On Quasi-Interpretations, Blind Abstractions and Implicit Complexity

Patrick Baillot, Ugo Dal Lago, Jean-Yves Moyen

▶ **To cite this version:**

Patrick Baillot, Ugo Dal Lago, Jean-Yves Moyen. On Quasi-Interpretations, Blind Abstractions and Implicit Complexity. 8th International Workshop on Logic and Computational Complexity (LCC'06), August 10 - 11, 2006 (Satellite Workshop of FLOC-LICS 2006), 2006, Seattle, United States. hal-00023668

HAL Id: hal-00023668
https://hal.science/hal-00023668

Submitted on 3 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

On Quasi-Interpretations, Blind Abstractions and Implicit Complexity*

Patrick Baillot[1], Ugo Dal Lago[1,2], and Jean-Yves Moyen[1]

[1] LIPN, Université Paris 13
[2] Università di Verona

Abstract. Quasi-interpretations are a technique to guarantee complexity bounds on first-order functional programs: with termination orderings they give in particular a sufficient condition for a program to be executable in polynomial time ([MM00]), called here the P-criterion. We study properties of the programs satisfying the P-criterion, in order to better understand its intensional expressive power.
Given a program on binary lists, its blind abstraction is the non-deterministic program obtained by replacing lists by their lengths (natural numbers). A program is blindly polynomial if its blind abstraction terminates in polynomial time. We show that all programs satisfying a variant of the P-criterion are in fact blindly polynomial. Then we give two extensions of the P-criterion: one by relaxing the termination ordering condition, and the other one (the bounded value property) giving a necessary and sufficient condition for a program to be polynomial time executable, with memoisation.

1 Introduction

Implicit computational complexity (ICC) explores machine-free characterizations of complexity classes, without referring to explicit resource bounds but instead by seeing these bounds as consequences of restrictions on program structures. It has been mainly developed in the functional programming paradigm, by taking advantage of ideas from primitice recursion ([Lei94,BC92]), proof-theory and linear logic ([Gir98]), rewriting systems ([BMM]), type systems ([Hof99])...

Usually ICC results include a soundness and a completeness statement: the first one says that all programs of a given language (or those satisfying a criterion) admit a certain complexity property, and the latter one that all *functions* of the corresponding functional complexity class can be programmed in this language. For instance in the case of polynomial time complexity the first statement refers to a polynomial time evaluation of programs, whereas the second, which is of an *extensional* nature, refers to the class FP of functions computable in polynomial time. Theorems of this kind have been given for many systems, like for instance ramified recursion [BC92], variants of linear logic, fragments of functional languages...

* Work partially supported by projects CRISS (ACI), NO-CoST (ANR)

Expressivity. This line of work is motivating from a programming language perspective, because it suggests ways to control complexity properties of programs, which is a difficult issue because of its infinitistic nature. However, extensional correspondence with complexity classes is usually not enough: a programming language (or a static analysis methodology for it) offering guarantees in terms of program safety is plausible only if it captures enough interesting and natural *algorithms*.

This issue has been pointed out by several authors ([MM00,Hof99]) and advances have been made in the direction of more liberal ICC systems: some examples are type systems for *non-size-increasing* computation and quasi-interpretations. However it is not always easy to measure the improvement a new ICC system brings up. Until now this has been usually illustrated by providing examples.

We think that to compare in a more appropriate way the algorithmic aspects of ICC systems and in particular to understand their limitations, specific methods should be developed. Indeed, what we need are *sharp results* on the *intensional* expressive power of existing systems and on the intrinsic limits of implicit complexity as a way to isolate large (but decidable) classes of programs with bounded complexity. For that we aim to establish properties (like necessary conditions) of the *programs* captured by an ICC system.

Quasi-interpretations (QI) can be considered as a static analysis methodology to infer asymptotic resource bounds for first-order functional programs. Used with termination orderings they allow to define various criteria to guarantee either space or time complexity bounds ([BMM,BMM05,Ama05,BMP06]). They present several advantages: the language for which they can be used is simple to use, and more importantly the class of programs captured by this approach is large compared to that caught by other ICC systems. Indeed, all primitive recursive programs from Bellantoni and Cook's function algebra can be easily proved to have a QI ([Moy03]). One particularly interesting criterion, that we will call here the *P-criterion*, says that programs with certain QIs and recursive path orderings can be evaluated in polynomial time.

In this paper, we focus our attention on QIs, proving a strong necessary condition for first-order functional programs on lists having a QI. More precisely, a program transformation called *blind abstraction* is presented. It consists in collapsing the constructors for lists to just one, modifying rewriting rules accordingly. This produces in general non-confluent programs, the efficiency of which can be evaluated by considering all possible evaluations of the program.

In general, the blind abstraction of a polytime first-order functional program on binary lists is not itself polytime: blinding introduces many paths that are not available in the original program. However, we show that under certain assumptions, blinding a program satisfying the P-criterion with a (uniform) QI always produces a program which is polytime, independently from non-confluence.

Outline. We first describe the syntax and operational semantics of programs (Section 2), before termination orderings and QIs (Section 3). Then blind abstractions are introduced (Section 4) and we give the main property of the

P-criterion w.r.t. blinding in section 5, with application to safe recursion. They lead us to study properties of particular classes of QIs in Section 6. Finally we define a generalization of the previous termination ordering and of QIs (bounded values property) which also guarantees the P-criterion (Section 7).

2 Programs as Term Rewriting Systems

2.1 Syntax and Semantics of Programs

We consider first-order term rewriting systems (TRS) with disjoint sets $\mathcal{X}$, $\mathcal{F}$, $\mathcal{C}$ resp. of variables, function symbols and constructor symbols.

Definition 1 (Syntax). *The sets of terms and the equations are defined by:*

$$\begin{array}{ll} \textit{(Constructor terms)} & \mathcal{T}(\mathcal{C}) \ni v ::= \mathbf{c} \mid \mathbf{c}(v_1, \cdots, v_n) \\ \textit{(terms)} & \mathcal{T}(\mathcal{C}, \mathcal{F}, \mathcal{X}) \ni t ::= \mathbf{c} \mid x \mid \mathbf{c}(t_1, \cdots, t_n) \mid \mathtt{f}(t_1, \cdots, t_n) \\ \textit{(patterns)} & \mathcal{P} \ni p ::= \mathbf{c} \mid x \mid \mathbf{c}(p_1, \cdots, p_n) \\ \textit{(equations)} & \mathcal{D} \ni d ::= \mathtt{f}(p_1, \cdots, p_n) \to t \end{array}$$

where $x \in \mathcal{X}$, $\mathtt{f} \in \mathcal{F}$, and $\mathbf{c} \in \mathcal{C}$. We shall use a type writer font for function symbols and a bold face font for constructors.

Definition 2 (Programs). *A program is a tuple $\mathtt{f} = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ where $\mathcal{E}$ is a set of $\mathcal{D}$-equations. Each variable in the right hand side (rhs) of an equation also appears in the lhs of the same equation. The program has a main function symbol in $\mathcal{F}$, which we shall also call $\mathtt{f}$.*

The domain of the computed functions is the constructor algebra $\mathcal{T}(\mathcal{C})$. A substitution σ is a mapping from variables to terms. We note $\mathfrak{S}$ the set of *constructor substitutions*, *i.e.* substitutions σ with range $\mathcal{T}(\mathcal{C})$.

Our programs are not necessarily deterministic, that is the TRS is not necessarily confluent. Non-confluent programs correspond to relations rather than functions. Notice that we could define a function by, *e.g.* , choosing the greatest possible result among all the executions [DP02]. Since we only consider the complexity (that is the length of all executions), this is not important here.

Firstly, we consider a call by value semantics which is displayed in Figure 1. The meaning of $t \downarrow v$ is that t evaluates to the constructor term v. A derivation will be called a *reduction proof*. The program $\mathtt{f}$ computes a relation $[\![\mathtt{f}]\!] : \mathcal{T}(\mathcal{C})^n \to \mathcal{T}(\mathcal{C})$ defined by: for all $u_i \in \mathcal{T}(\mathcal{C}), v \in [\![\mathtt{f}]\!](u_1, \cdots, u_n)$ iff there is a derivation for $\mathtt{f}(u_1, \cdots, u_n) \downarrow v$.

The size $|J|$ of a judgement $J = t \downarrow v$ is the size $|t|$ of the lhs term.

Definition 3 (Active rules). Passive *semantics rules are Constructor and Split. The only* active *rule is Function.*

Let $\pi : t \downarrow v$ be a reduction proof. If we have:

$$\frac{e = \mathtt{g}(q_1, \cdots, q_n) \to r \in \mathcal{E} \quad \sigma \in \mathfrak{S} \quad q_i\sigma = u_i \quad r\sigma \downarrow u}{s = \mathtt{g}(u_1, \cdots, u_n) \downarrow u} \textit{(Function)}$$

$$\frac{\mathbf{c} \in \mathcal{C} \quad t_i \downarrow v_i}{\mathbf{c}(t_1, \cdots, t_n) \downarrow \mathbf{c}(v_1, \cdots, v_n)}\ (\text{Constructor}) \qquad \frac{\exists j, t_j \notin \mathcal{T}(\mathcal{C}) \quad t_i \downarrow v_i \quad \mathtt{f}(v_1, \cdots, v_n) \downarrow v}{\mathtt{f}(t_1, \cdots, t_n) \downarrow v}\ (\text{Split})$$

$$\frac{\mathtt{f}(p_1, \cdots, p_n) \to r \in \mathcal{E} \quad \sigma \in \mathfrak{S} \quad p_i\sigma = v_i \quad r\sigma \downarrow v}{\mathtt{f}(v_1, \cdots, v_n) \downarrow v}\ (\text{Function})$$

Fig. 1. Call by value semantics with respect to a program $\langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$

then we say that term s (resp. judgment $J = s \downarrow u$) is active. *e is the equation* activated *by s (resp. J) and $r\sigma$ (resp. $r\sigma \downarrow u$) is the* activation *of s (resp. J). Other judgements (conclusions of (Split) or (Constructor) rules) are called passive.*

Notice that the set of active terms in a proof π is exactly the set of terms of the form $\mathtt{f}(v_1, \cdots, v_n)$, where v_i are constructor terms, appearing in π. Since the program may be non deterministic, the equation activated by a term s depends on the reduction and on the occurrence of s in π, and not only on s.

Lemma 4. *For each program, there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$ such that for any reduction proof π, active term t in it and s the activation of t, $|s| = p(|t|)$. That is, the size of s is polynomially bounded by the size of t.*

Proof. Because there is only a finite set of equations in a program, each of them leading to at most a polynomial increase in size.

A subderivation of a derivation proof is obtained by recursively taking a judgement and some of its premises. That is, it is a subgraph of the proof tree that is also a tree (but does not necessarily go down to the leaves of the proof).

Definition 5 (Dependences). *Let π be a reduction proof and $J = t \downarrow v$ be a passive judgement appearing in it. A* dependence *of J is a subderivation of π whose root is J and which contains only passive judgements. The maximum dependence D_J is the biggest dependence of a judgement, with respect to inclusion.*

The following is an example program that we will use throughout the paper:

$$\begin{array}{lll} \mathtt{f}(\mathbf{s}_0\mathbf{s}_i x) & \to \mathtt{append}(\mathtt{f}(\mathbf{s}_1 x), \mathtt{f}(\mathbf{s}_1 x)) & \text{for } i = \{0,1\}, \\ \mathtt{f}(\mathbf{s}_1 x) & \to x, & \mathtt{f}(\mathbf{nil}) \to \mathbf{nil} \\ \mathtt{append}(\mathbf{s}_i x, y) & \to \mathbf{s}_i \mathtt{append}(x, y) \quad \text{for } i \in \{0,1\}, & \mathtt{append}(\mathbf{nil}, y) \to y \end{array}$$

Consider the derivation π from Figure 2.

$$\dfrac{\dfrac{\dfrac{\mathbf{nil} \downarrow \mathbf{nil}}{\mathtt{f}(\mathbf{s}_1\mathbf{nil}) \downarrow \mathbf{nil}} \quad \dfrac{\mathbf{nil} \downarrow \mathbf{nil}}{\mathtt{f}(\mathbf{s}_1\mathbf{nil}) \downarrow \mathbf{nil}} \quad \dfrac{\mathbf{nil} \downarrow \mathbf{nil}}{\mathtt{append}(\mathbf{nil}, \mathbf{nil}) \downarrow \mathbf{nil}}}{\mathtt{append}(\mathtt{f}(\mathbf{s}_1\mathbf{nil}), \mathtt{f}(\mathbf{s}_1\mathbf{nil})) \downarrow \mathbf{nil}}}{\mathtt{f}(\mathbf{s}_0\mathbf{s}_1\mathbf{nil}) \downarrow \mathbf{nil}}$$

Fig. 2. Example of reduction proof.

Active judgements appearing in π are $\mathtt{f}(\mathbf{s}_0\mathbf{s}_1\mathbf{nil}) \downarrow \mathbf{nil}$, $\mathtt{f}(\mathbf{s}_1\mathbf{nil}) \downarrow \mathbf{nil}$ and $\mathtt{append}(\mathbf{nil}, \mathbf{nil}) \downarrow \mathbf{nil}$. Let $J = \mathtt{append}(\mathtt{f}(\mathbf{s}_1\mathbf{nil}), \mathtt{f}(\mathbf{s}_1\mathbf{nil})) \downarrow \mathbf{nil}$ be a passive judgement. Its maximum dependence D_J is the subderivation containing J itself.

Lemma 6. *Let π be a reduction proof and $J = t \downarrow v$ be a judgement in it.*

1. *The depth of any dependence of J is bounded by the depth of t.*
2. *For each judgement $s \downarrow u$ in a dependence of J, s is a subterm of t.*
3. *The number of judgements in a dependence of J is bounded by the size of t.*

Proof. We can either do a quick induction or look at the rules.

1. Because each passive rule decreases the depth of the term.
2. Because passive rules only produce proper subterms or active terms.
3. It is the number of subterms of t.

Proposition 7. *For all programs, there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$ such that for all derivations π and all active judgement J in it with a passive activation H, $|D_H| \leq p(|J|)$. That is, the size of D_H is polynomially bounded by the size of J and the polynomial is only dependent on the program.*

Proof. Obtained by combining the results of Lemmas 4 and 6.

Proposition 8. *For all programs, there exists a polynomial $p : \mathbb{N}^2 \to \mathbb{N}$ such that for all derivation proof π, if A is the number of active judgements in π and S is the maximum size of an active judgement then $|\pi| \leq p(A, S)$.*

So, to bound the time of a derivation, it is sufficient to bound the number and size of active terms, passive terms playing no real role in it.

Next, we also consider a call by value semantics with memoisation for confluent programs. The idea is to maintain a cache to avoid recomputing the same things several times. Each time a function call is performed, the semantics looks in the cache. If the same call has already been computed, then the result can be given immediately, else, we need to compute the corresponding value and store the result in the cache (for later reuse). The memoisation semantics is displayed on Figure 3. The (Update) rule can only be triggered if the (Read) rule cannot, that is if there is no v such that $(\mathtt{f}, v_1, \cdots, v_n, v) \in C$.

Memoisation corresponds to an automation of the algorithmic technique of dynamic programming.

$$\frac{\mathbf{c} \in \mathcal{C} \quad \langle C_{i-1}, t_i\rangle \Downarrow \langle C_i, v_i\rangle}{\langle C_0, \mathbf{c}(t_1, \cdots, t_n)\rangle \Downarrow \langle C_n, \mathbf{c}(v_1, \cdots, v_n)\rangle}\ (\text{Constructor})$$

$$\frac{\exists j, t_j \notin \mathcal{T}(\mathcal{C}) \quad \langle C_{i-1}, t_i\rangle \Downarrow \langle C_i, v_i\rangle \quad \langle C_n, \mathtt{f}(v_1, \cdots, v_n)\rangle \Downarrow \langle C, v\rangle}{\langle C_0, \mathtt{f}(t_1, \cdots, t_n)\rangle \Downarrow \langle C, v\rangle}\ (\text{Split})$$

$$\frac{(\mathtt{f}, v_1, \cdots, v_n, v) \in C}{\langle C, \mathtt{f}(v_1, \cdots, v_n)\rangle \Downarrow \langle C, v\rangle}\ (\text{Read})$$

$$\frac{\mathtt{f}(p_1, \cdots, p_n) \to r \in \mathcal{E} \quad \sigma \in \mathfrak{S} \quad p_i\sigma = v_i \quad \langle C, r\sigma\rangle \Downarrow \langle C', v\rangle}{\langle C, \mathtt{f}(v_1, \cdots, v_n)\rangle \Downarrow \langle C' \cup (\mathtt{f}, v_1, \cdots, v_n, v), v\rangle}\ (\text{Update})$$

Fig. 3. Call-by-value interpreter with Cache of $\langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E}\rangle$.

The expression $\langle C, t\rangle \Downarrow \langle C', v\rangle$ means that the computation of t is v, given a program `f` and an initial cache C. The final cache C' contains C and each call which has been necessary to complete the computation.

Definition 9 (Active rules). *Constructor and Split are* passive *semantic rules. Update is an* active *rule. Read is a* semi-active *rule.*

Definition 10 (Active terms). *Active terms and judgements, activated equations, activations and dependences are defined similarly as for the call by value case. Semi-active terms and judgements are similarly defined for semi-active rules.*

Lemma 6 still holds, but we also need to bound the number of semi-active judgements to bound the size of a derivation. However, since they only lead to leaves in the derivation and the arity of the derivation tree is bounded (by k the maximum arity of a symbol in $\mathcal{F} \bigcup \mathcal{C}$), there is at most k times more (Read) rule than the total number of other rules.

Proposition 11. *Consider the memoisation semantics of Figure 3. For all programs, there exists a polynomial P such that for all derivation proof π, if A is the number of active judgements in π and S is the maximum size of an active judgement then $|\pi| \leq P(A, S)$.*

Proof. Proposition 7 still allows to bound the size of dependences, hence the number of passive judgements. Since semi-active judgements form a subset of the set of leaves in π and since the number of premises of a rule is statically bounded (by k, the maximum arity of a symbol), the number of semi-active judgements is polynomially bounded by the number of non-leaves judgements in π, hence by the number of active and passive judgements.

Lemma 12. *Let $J_1 = \langle C_1, t\rangle \Downarrow \langle C_1, v\rangle$ be a semi-active judgement in a proof $\pi : \langle \emptyset, t\rangle \Downarrow \langle C, v\rangle$, then there exists an active judgement $J_2 = \langle C_2, t\rangle \Downarrow \langle C_2', v\rangle$ in π.*

Proof. Because the couple can only be in the cache if an active judgement put it there.

The naive model where each rules takes unary time to be executed is not very realistic with the memoisation semantics. Indeed, each (Read) and (Update) rule needs to perform a lookup in the cache and this would take time proportional to the size of the cache (and the size of elements in it). However, the size of the final cache is exactly the number of (Update) rules in the proof (because only (Update) modify the cache) and the size of terms in the cache is bounded by the size of active terms (only active terms are stored in the cache). So Proposition 11 yields to a polynomial bound on the execution time.

Memoisation cannot be used with non-confluent programs. Indeed, the same function call can lead to several different results. Several ideas could be used to define a memoisation semantics for non-confluent programs, but they all have

their problems, hence we won't use any of them here and only use memoisation when the program is confluent. For sufficient conditions to decide if a program is confluent or not, refer, typically, to Huet's work [Hue80]. Here are, nevertheless, several different hints on how to design a memoisation semantics for non-confluent programs.

- (No lookup): The cache is never used and everything is recomputed every time. This is clearly not satisfactory since this is exactly the same thing as the cbv semantics.
- (Cache first): If a function call is in the cache, use it. This is clearly not satisfactory because two identical calls will lead to the same result even if there was some non confluence involved.
- (Random lookup): When performing a call, randomly choose between using a result in the cache and doing the computation. This is not satisfactory because we can choose to always recompute things, hence exactly mimicking the cbv semantics and the worst case will be the same (no time is gained).
- (Random lookup with penalties): Same as random lookup, but after (re)computing a function call, check if is was already in the cache. If so, abort (because one should have looked for the result in the cache rather than recomputing it). This seems rather satisfactory but brings in lots of problems for analysis. In particular, calls following different paths but leading to the same result will be identified even if they shouldn't.

2.2 Call trees, call dags

Following [BMM], we present now call-trees which are a tool that we shall use all along. Let $\mathtt{f} = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ be a program. A call-tree gives a static view of an execution and captures all function calls. Hence, we can study dependencies between function calls without taking care of the extra details provided by the underlying rewriting relation.

Definition 13 (States). *A* state *is a tuple* $\langle \mathtt{f}, v_1, \cdots, v_n \rangle$ *where* $\mathtt{f}$ *is a function symbol of arity* n *and* $v_1, \cdots, v_n$ *are constructor terms. Assume that* $\eta_1 = \langle \mathtt{f}, v_1, \cdots, v_n \rangle$ *and* $\eta_2 = \langle \mathtt{g}, u_1, \cdots, u_m \rangle$ *are two states. A* transition *is a triplet* $\eta_1 \overset{e}{\rightsquigarrow} \eta_2$ *such that:*

1. *e is an equation* $\mathtt{f}(p_1, \cdots, p_n) \to t$ *of* $\mathcal{E}$,
2. *there is a substitution* σ *such that* $p_i\sigma = v_i$ *for all* $1 \leq i \leq n$,
3. *there is a subterm* $\mathtt{g}(s_1, \cdots, s_m)$ *of* t *such that* $s_i\sigma \downarrow u_i$ *for all* $1 \leq i \leq m$.

$\overset{*}{\rightsquigarrow}$ *is the reflexive transitive closure of* $\cup_{e \in \mathcal{E}} \overset{e}{\rightsquigarrow}$.

Definition 14 (Call trees). *Let* $\pi : t \downarrow v$ *be a reduction proof. Its* call trees *is the set of tree* Θ_π *obtained by only keeping active terms in* π.

That is, if t is passive:

$$\frac{b \in \mathcal{F} \bigcup \mathcal{C} \quad \pi_i : t_i \downarrow v_i}{b(t_1, \cdots, t_n) \downarrow b(v_1, \cdots, v_n)} \text{ (C) or (S)}$$

Then $\Theta_\pi = \bigcup \Theta_{\pi_i}$.

If t is active:

$$\frac{\mathtt{f}(p_1, \cdots, p_n) \to r \in \mathcal{E} \quad \sigma \in \mathfrak{S} \quad p_i\sigma = v_i \quad \rho : r\sigma \downarrow v}{\mathtt{f}(v_1, \cdots, v_n) \downarrow v} \text{ (F)}$$

Then Θ_π only contains the tree whose root is $\langle \mathtt{f}, v_1, \cdots, v_n \rangle$ and children are Θ_ρ.

When using the semantics with memoisation, the call-dag of a state is defined similarly to the call-tree, but using a direct acyclic graph instead of a tree, that is by adding links from (Read)-judgements to the corresponding (Update)-judgement. Notice that (Read)-judgement are always leaves of the proof so we do not loose any part of the proof by doing so.

Definition 15 (Call dags). *Let $\pi : \langle C, t \rangle \Downarrow \langle C', v \rangle$ be a reduction proof. Its* call trees *is the set of trees obtained by keeping only active terms and its* call dag *Θ_π is obtained by keeping only active terms and replacing each semi-active term by a link to the corresponding (via Lemma 12) active term.*

Fact 1 (call tree arity) *Let f be a program. There exists a fixed integer k, such that given a derivation π of a term of the program, and a tree $\mathcal{T}$ of Θ_π, all nodes in $\mathcal{T}$ have at most k sons.*

Proof. For each r.h.s. term r of an equation of f consider the number of maximal subterms of r with a function as head symbol; let then k be the maximum of these integers over the (finite) set of equations of f.

Lemma 16. *Let π be a proof, $\mathcal{T}$ be a call-tree (call-dag) in Θ_π and consider two states $\eta = \langle \mathtt{f}, v_1, \cdots, v_n \rangle$ and $\eta' = \langle \mathtt{g}, u_1, \cdots, u_m \rangle$ such that η' is a child of η. Let $t = \mathtt{f}(v_1, \cdots, v_n)$ be the active term corresponding to η and $s = \mathtt{g}(u_1, \cdots, u_m)$ be the active term corresponding to η'. Let e be the equation activated by t in π. Then, $\eta \overset{e}{\rightsquigarrow} \eta'$.*

Conversely, if $\eta \overset{e}{\rightsquigarrow} \eta'$ and e is activated by t then η' is a child of η in $\mathcal{T}$.

Proof. Condition 1 and 2 correspond to the application of the active rule. Condition 3 correspond to the application of several passive rules to get ride of the context and evaluate the parameters of $\mathtt{g}$.

This means that our definition of call trees is equivalent to the one in [BMM]. However, we need an alternate definition in order to deal with non determinism.

Call trees and call dags are a tool to easily count the number of active judgements in a derivation. So, now, in order to apply Propositions 8 or 11 we need to (i) bound the size (number of nodes) in the call tree or call dag and (ii) bound the size of the states appearing in the call tree (dag).

3 Ordering, Quasi-Interpretations

3.1 Termination Orderings

Definition 17 (Precedence). *Let $\mathcal{f} = \langle \mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E} \rangle$ be a program. A* precedence *$\preceq_{\mathcal{F}}$ is a partial ordering over $\mathcal{F} \bigcup \mathcal{C}$. We note $\approx_{\mathcal{F}}$ the associated equivalence relation. A precedence is* compatible *with $\mathcal{f}$ if for each equation $\mathtt{f}(p_1, \cdots, p_n) \to r$ and each symbol b appearing in r, $b \preceq_{\mathcal{F}} \mathtt{f}$. It is* separating *if for each $\mathbf{c} \in \mathcal{C}, \mathtt{f} \in \mathcal{F}$, $\mathbf{c} \prec_{\mathcal{F}} \mathtt{f}$ (that is constructors are the smallest elements of $\prec_{\mathcal{F}}$ while functions are the biggest). It is* fair *is for each constructors $\mathbf{c}, \mathbf{c}'$ with the same arity, $\mathbf{c} \approx_{\mathcal{F}} \mathbf{c}'$ and it is* strict *if for each constructors $\mathbf{c}, \mathbf{c}'$, $\mathbf{c}$ and $\mathbf{c}'$ are incomparable.*

Any strict precedence can be canonically extended into a fair precedence.

Definition 18 (Product extension). *Let $\prec$ be an ordering over a set S. Its* product extension *is an ordering $\prec^p$ over tuples of elements of S such that $(m_1, \cdots, m_k) \prec^p (n_1, \cdots, n_k)$ if and only if (i) $\forall i, m_i \preceq n_i$ and (ii) $\exists j$ such that $m_j \prec n_j$.*

Definition 19 (PPO). *Given a separating precedence $\preceq_{\mathcal{F}}$, the recursive path ordering $\prec_{rpo}$ is defined in Figure 4.*

If $\prec_{\mathcal{F}}$ is strict (resp. fair) and separating, then the ordering is the Product Path Ordering *PPO (resp. the* extended Product Path Ordering *EPPO).*

Of course, it is possible to consider other extensions of orderings. Usual choices are the lexicographic extension, thus leading to Lexicographic Path Ordering or Multiset extension, leading to Multiset path Ordering. It is also possible to add a notion of *status* to function [KL80] indicating with which extension the parameters must be compared. This leads to the more general Recursive Path Ordering (RPO). However, here we only use the (extended) Product Path Ordering so we don't describe others.

$$\frac{s = t_i \text{ or } s \prec_{rpo} t_i}{s \prec_{rpo} \mathtt{f}(\ldots, t_i, \ldots)}\, \mathtt{f} \in \mathcal{F} \bigcup \mathcal{C} \qquad \frac{\forall i\ s_i \prec_{rpo} \mathtt{f}(t_1, \cdots, t_n) \qquad \mathtt{g} \prec_{\mathcal{F}} \mathtt{f}}{g(s_1, \cdots, s_m) \prec_{rpo} \mathtt{f}(t_1, \cdots, t_n)}\, \mathtt{f}, \mathtt{g} \in \mathcal{F} \bigcup \mathcal{C}$$

$$\frac{(s_1, \cdots, s_n) \prec^p_{rpo} (t_1, \cdots, t_n) \qquad \mathtt{f} \approx_{\mathcal{F}} \mathtt{g} \qquad \forall i\ s_i \prec_{rpo} \mathtt{f}(t_1, \cdots, t_n)}{\mathtt{g}(s_1, \cdots, s_n) \prec_{rpo} \mathtt{f}(t_1, \cdots, t_n)}\, \mathtt{f}, \mathtt{g} \in \mathcal{F} \bigcup \mathcal{C}$$

Fig. 4. Definition of $\prec_{rpo}$

An equation $l \to r$ is decreasing if we have $r \prec_{rpo} l$. A program is ordered by $\prec_{rpo}$ if there is a separating precedence on $\mathcal{F}$ such that each equation is decreasing. Recall that $\prec_{rpo}$ guarantees termination ([Der82]).

Notice that in our case, since patterns cannot contain function symbols, if there is a precedence such that the program is ordered by the corresponding $\prec_{rpo}$, then there is also a compatible one with the same condition.

Lemma 20. *Let $\mathtt{f}$ be a program and $\prec_{\mathcal{F}}$ be a separating precedence compatible with it. Let $\eta = \langle \mathtt{f}, v_1, \cdots, v_n \rangle$ be a state in a call tree (resp. dag) $\mathcal{T}$ and $\eta' = \langle \mathtt{g}, u_1, \cdots, u_m \rangle$ be a descendant of η in $\mathcal{T}$. Then $\mathtt{g} \preceq_{\mathcal{F}} \mathtt{f}$*

Proof. Because the precedence is compatible with $\mathtt{f}$.

Proposition 21 (Computing by rank). *Let $\mathtt{f}$ be a program and $\prec_{\mathcal{F}}$ be a separating precedence compatible with it. Let $\mathcal{T}$ be a call tree (resp. dag) and $\eta = \langle \mathtt{f}, v_1, \cdots, v_n \rangle$ be a node in it. Let A be the maximum number of descendants of a node with the same arity:*

$$A = \max_{\eta = \langle \mathtt{f}, v_1, \cdots, v_n \rangle \in \mathcal{T}} \#\{\eta' = \langle \mathtt{g}, u_1, \cdots, u_m \rangle, \eta \text{ is an ancestor of } \eta' \text{ and } \mathtt{g} \approx_{\mathcal{F}} \mathtt{f}\}$$

The size of $\mathcal{T}$ is polynomially bounded by A.

Proof. Let $\mathtt{f}$ be a function symbol. Its rank is $rk(\mathtt{f}) = \max_{\mathtt{g} \prec_{\mathcal{F}} \mathtt{f}} rk(\mathtt{g}) + 1$.

Let d be the maximum number of function symbols in a rhs of $\mathtt{f}$ and k be the maximum rank. We will prove by induction that there are at most $B_i = \sum_{i \leq j \leq k} d^{k-j} \times A^{k-j+1}$ nodes in $\mathcal{T}$ at rank i.

The root has rank k. Hence, there are at most $A = d^{k-k} A^{k-k+1} = B_k$ nodes at rank k.

Suppose that the hypothesis is true for all ranks $j > i$. Each node has at most d children. Hence, there are at most $d \sum_{j>i} B_j$ nodes at rank i whose parent has rank $\neq i$. Each of these nodes has at most A descendants at rank i, hence there are at most $d \times A \times \sum_{j>i} B_j < B_i$ nodes at rank i.

Since $B_i < (k - i + 1) \times d^k A^{k+1}$, $\sum B_i$ is polynomially bounded in A and so is the size of the call tree (dag).

Thus to bound the number of active rules in a derivation (hence bound the derivation's size by Prop. 8 or 11) it suffices to establish the bound rank by rank.

Proposition 22. *Let $\mathtt{f}$ be a program terminating by PPO, $\mathcal{T}$ be a call dag and $\eta = \langle \mathtt{f}, v_1, \cdots, v_n \rangle$ be a node in $\mathcal{T}$. The number of descendants of η in $\mathcal{T}$ with the same rank as η is polynomially bounded by $|\eta|$.*

Proof. Because of the termination ordering, if $\eta' = \langle \mathtt{g}, u_1, \cdots, u_m \rangle$ is a descendant of η with $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}$, then u_i is a subterm of v_i. There are at most $|v_i|$ such subterms and thus $c\Pi(|v_i|+1)$ possible nodes (where c is the number of functions with the same precedence as $\mathtt{f}$).

This is point (2) in the proof of Lemma 51 in [BMM]. Notice that it only works on a call dag, because identical nodes are identified, and not on a call tree.

3.2 Quasi-interpretations

We restrict ourselves to additive QIs as defined in [BMM].

Definition 23 (Assignment). *An* assignment *of a symbol $b \in \mathcal{F} \bigcup \mathcal{C}$ whose arity is n is a function $(\!|b|\!) : (\mathbb{R})^n \to \mathbb{R}$ such that:*

(Subterm) $(\!|b|\!)(X_1, \cdots, X_n) \geq X_i$ *for all* $1 \leq i \leq n$.
(Weak Monotonicity) $(\!|b|\!)$ *is increasing (not strictly) wrt each variable.*
(Additivity) $(\!|\mathbf{c}|\!)(X_1, \cdots, X_n) \geq \sum_{i=1}^{n} X_i + a$ *if* $\mathbf{c} \in \mathcal{C}$ *(where* $a \geq 1$*).*
(Polynomial) $(\!|b|\!)$ *is bounded by a polynomial.*

We extend assignments $(\!|.|\!)$ to terms canonically. Given a term t with n variables, the assignment $(\!|t|\!)$ is a function $(\mathbb{R})^n \to \mathbb{R}$ defined by the rules:

$$(\!|b(t_1, \cdots, t_n)|\!) = (\!|b|\!)((\!|t_1|\!), \cdots, (\!|t_n|\!))$$
$$(\!|x|\!) = X$$

Given two functions $f : (\mathbb{R})^n \to \mathbb{R}$ and $g : (\mathbb{R})^m \to \mathbb{R}$ such that $n \geq m$, we say that $f \geq g$ iff $\forall X_1, \ldots, X_n : f(X_1, \ldots, X_n) \geq g(X_1, \ldots, X_m)$.

There are some well-known and useful consequences of such definitions. We have $(\!|s|\!) \geq (\!|t|\!)$ if t is a subterm of s. Then, for every substitution σ, $(\!|s|\!) \geq (\!|t|\!)$ implies that $(\!|s\sigma|\!) \geq (\!|t\sigma|\!)$.

Definition 24 (Quasi-interpretation). *A program assignment $(\!|.|\!)$ is an assignment of each program symbol. An assignment $(\!|.|\!)$ of a program is a quasi-interpretation (QI) if for each equation $l \to r$,*

$$(\!|l|\!) \geq (\!|r|\!).$$

In the following, unless explicitly specified, $(\!|.|\!)$ will always denote a QI and not an assignment.

Lemma 25. *Let v be a constructor term, $|v| \leq (\!|v|\!) \leq a|v|$ for a constant a.*

Proof. By induction. the constant a depends on the constants in the QI of constructors.

Lemma 26. *Assume f has a QI. Let $\eta = \langle \mathtt{f}, v_1, \cdots, v_n \rangle$ and $\eta' = \langle \mathtt{g}, u_1, \cdots, u_m \rangle$ be two states such that $\eta \overset{*}{\rightsquigarrow} \eta'$. Then, $(\!|\mathtt{g}(u_1, \cdots, u_m)|\!) \leq (\!|\mathtt{f}(v_1, \cdots, v_n)|\!)$.*

Proof. Because $\mathtt{g}(u_1, \cdots, u_m)$ is a subterm of a term obtained by reduction from $\mathtt{f}(v_1, \cdots, v_n)$.

Corollary 27. *Let f be a program admitting a QI and $\pi : \langle C, t = \mathtt{f}(v_1, \cdots, v_n) \rangle \Downarrow \langle C', v \rangle$ be a derivation. The size of any active term in π is bounded by $P(|v_i|)$ for a given polynomial P.*

Proof. The size of an active term $s = \mathtt{g}(u_1, \cdots, u_m)$ is bounded by $m \max |u_i| \leq m(\!|s|\!)$. By the previous Lemma, $(\!|s|\!) \leq (\!|t|\!)$. But by polynomiality of QIs, $(\!|t|\!) \leq Q((\!|v_i|\!))$. Since v_i are constructor terms, $(\!|v_i|\!) \leq a|v_i|$

Now, if we combine this bound on the size of active terms together with the bound on the number of active terms of Proposition 22, we can apply Proposition 11 and conclude that programs terminating by PPO and admitting a QI are PTIME computable. Actually, the converse is also true:

Theorem 28 (P-criterion, (Bonfante, Marion, Moyen [BMM])). *The set of functions computables by programs that (i) terminates by PPO and (ii) admits a QI is exactly* PTIME.

In order to achieve the polynomial bound, it is necessary to use the cbv semantics with memoisation.

4 Blind Abstractions of Programs

4.1 Definitions

Our idea is to associate to a given program $\mathtt{f}$ an abstract program $\overline{\mathtt{f}}$ obtained by forgetting each piece of data and replacing it by its size as a unary integer. In this way, even if $\mathtt{f}$ is deterministic, the associated $\overline{\mathtt{f}}$ will in general not be deterministic.

For that we first define a target language:

- variables: $\overline{\mathcal{X}} = \mathcal{X}$,
- function symbols: $\overline{\mathcal{F}} = \{\overline{\mathtt{f}}, \mathtt{f} \in \mathcal{F}\}$,
- constructor symbols: $\overline{\mathcal{C}} = \{\mathbf{s}, \mathbf{0}\}$ where $\mathbf{s}$ (resp. $\mathbf{0}$) has arity 1 (resp. 0).

This language defines a set of constructor terms $\mathcal{T}(\overline{\mathcal{C}})$, a set of terms $\mathcal{T}(\overline{\mathcal{C}}, \overline{\mathcal{F}}, \overline{\mathcal{X}})$ and a set of patterns $\overline{\mathcal{P}}$.

The *blinding map* is the natural map $\mathcal{B} : \mathcal{T}(\mathcal{C}, \mathcal{F}, \mathcal{X}) \longrightarrow \mathcal{T}(\overline{\mathcal{C}}, \overline{\mathcal{F}}, \overline{\mathcal{X}})$ obtained by replacing constructors of arity 1 with $\mathbf{s}$, and those of arity 0 by $\mathbf{0}$. It induces similar maps on constructor terms and patterns. We will write $\overline{t}$ (resp. $\overline{p}$) for $\mathcal{B}(t)$ (resp. $\mathcal{B}(p)$).

The blinding map extends to *equations* in the expected way: given a equation $d = p \to t$ of the language $(\mathcal{X}, \mathcal{F}, \mathcal{C})$, we set $\overline{d} = \mathcal{B}(d) = \overline{p} \to \overline{t}$. Finally, given a program $\mathtt{f} = (\mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E})$, its blind image is $\overline{\mathtt{f}} = (\overline{\mathcal{X}}, \overline{\mathcal{C}}, \overline{\mathcal{F}}, \overline{\mathcal{E}})$ where the equations are obtained by: $\overline{\mathcal{E}} = \{\overline{d}, d \in \mathcal{E}\}$. Observe that even if $\mathtt{f}$ is an orthogonal program, this will not necessarily be the case of $\overline{\mathtt{f}}$, because some patterns are identified by $\mathcal{B}$.

The denotational semantics of $\overline{\mathtt{f}}$ can be seen as a relation over the domain of tally integers.

4.2 Complexity Definitions

Definition 29 (Strongly polynomial). *We say a non-deterministic program* $\mathtt{f}$ *(of arity* n*) is strongly polynomial if there exists a polynomial* $p : \mathbb{N} \to \mathbb{N}$ *such that for every sequence* $v_1, \cdots, v_n$ *and any* $\pi : \mathtt{f}(v_1, \cdots, v_n) \downarrow u$*, it holds that* $|\pi| \leq p(\sum_{i=1}^{n} |v_i|)$.

Of course similar definitions would also make sense for other complexity bounds than polynomial. In the case of a deterministic program, this definition coincides with that of a polynomial time program (in the model where rewriting steps are counted as unit step).

Definition 30 (Blindly polynomial). *A program* $\mathtt{f}$ *is blindly polynomial if its blind abstraction* $\overline{\mathtt{f}}$ *is strongly polynomial.*

Observe that:

Fact 2 *If a program* $\mathtt{f}$ *is blindly polynomial, then it is polynomial time (with the call-by-value semantics).*

Indeed, it is sufficient to observe that any reduction sequence of $\mathtt{f}$ can be mapped by $\mathcal{B}$ to a reduction sequence of $\overline{\mathtt{f}}$. The converse property is not true. Observe for that our running example in Figure 5: note that $\mathtt{f}$ terminates in polynomial time but this is not the case for $\overline{\mathtt{f}}$. Indeed if we denote $\underline{n} = \underbrace{\mathbf{s} \dots \mathbf{s}}_{n}\ \mathbf{0}$, we have that $\overline{\mathtt{f}}(\underline{n})$ can be reduced in an exponential number of steps, with a $\pi : \overline{\mathtt{f}}(\underline{n}) \downarrow \underline{2^n}$.

$\mathtt{f}$			$\overline{\mathtt{f}}$		
$\mathtt{f}(\mathbf{s}_0\mathbf{s}_i x)$	$\to$	$\mathtt{append}(\mathtt{f}(\mathbf{s}_1 x), \mathtt{f}(\mathbf{s}_1 x))$	$\overline{\mathtt{f}}(\mathbf{s}\mathbf{s}x)$	$\to$	$\overline{\mathtt{append}}(\overline{\mathtt{f}}(\mathbf{s}x), \overline{\mathtt{f}}(\mathbf{s}x))$
$\mathtt{f}(\mathbf{s}_1 x)$	$\to$	x	$\overline{\mathtt{f}}(\mathbf{s}x)$	$\to$	x
$\mathtt{f}(\mathbf{nil})$	$\to$	$\mathbf{nil}$	$\overline{\mathtt{f}}(0)$	$\to$	0
$\mathtt{append}(\mathbf{s}_i x, y)$	$\to$	$\mathbf{s}_i \mathtt{append}(x, y)$	$\overline{\mathtt{append}}(\mathbf{s}x, y)$	$\to$	$\mathbf{s}\ \overline{\mathtt{append}}(x, y)$
$\mathtt{append}(\mathbf{nil}, y)$	$\to$	y	$\overline{\mathtt{append}}(0, y)$	$\to$	y

Fig. 5. Blind abstraction of our running example

Note that the property of being blindly polynomial is indeed a strong condition, because it means in some sense that the program will terminate with a polynomial bound for reasons which are indifferent to the actual content of the input but only depend on its size.

Now we want to discuss the behavior of the blinding map with respect to criteria on TRS based on recursive path orderings (RPO) and QIs ([BMM]).

4.3 Blinding and Recursive Path Orderings

Lemma 31. *Let* $\mathtt{f}$ *be a program: if* $\mathtt{f}$ *terminates by PPO then* $\overline{\mathtt{f}}$ *terminates by PPO.*

Indeed $\mathcal{F}$ and $\overline{\mathcal{F}}$ are in one-one correspondence, and it is easy to observe that: if $\prec_{\mathcal{F}}$ is a precedence which gives a PPO ordering for $\mathtt{f}$, then the corresponding $\prec_{\overline{\mathcal{F}}}$ does the same for $\overline{\mathtt{f}}$.

The converse is not true, see for example Figure 5 where the first equation does terminate by PPO on the blind side but not on the non-blind side. However, we have:

Proposition 32. *Let f be a program. The three following statements are equivalent: (i) $\mathtt{f}$ terminates by EPPO, (ii) $\overline{\mathtt{f}}$ terminates by EPPO, (iii) $\overline{\mathtt{f}}$ terminates by PPO.*

Proof. Just observe that on $\mathcal{T}(\overline{\mathcal{C}}, \overline{\mathcal{F}}, \overline{\mathcal{X}})$, PPO and EPPO coincide, and that $\overline{t} \prec_{EPPO} \overline{t'}$ implies $t \prec_{EPPO} t'$.

4.4 Blinding and Quasi-interpretations

Assume the program $\mathtt{f}$ admits a quasi-interpretation $(\!|.|\!)$. Then in general this does not imply that $\overline{\mathtt{f}}$ admits a quasi-interpretation. Indeed one reason why $(\!|.|\!)$ cannot be simply converted into a quasi-interpretation for $\overline{\mathtt{f}}$ is because a quasi-interpretation might in general give different assignments to several constructors of the same arity, for instance when $(\!|\mathbf{s}_0|\!)(X) = X + 1$ and $(\!|\mathbf{s}_1|\!)(X) = X + 2$. Then when considering $\overline{\mathtt{f}}$ there is no natural choice for $(\!|\mathbf{s}|\!)$.

However, in most examples in practice, a restricted class of quasi-interpretations is used:

Definition 33 (Uniform assignments). *An assignment for $\mathtt{f} = (\mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E})$ is uniform if all constructors of same arity have the same assignment: for each $\mathbf{c}, \mathbf{c}' \in \mathcal{C}$, $arity(\mathbf{c}) = arity(\mathbf{c}')$ implies $(\!|\mathbf{c}|\!) = (\!|\mathbf{c}'|\!)$. A quasi-interpretation of $\mathtt{f}$ is uniform if it is defined by a uniform assignment.*

Now we have:

Proposition 34. *The program $\mathtt{f}$ admits a uniform quasi-interpretation iff $\overline{\mathtt{f}}$ admits a quasi-interpretation.*

5 Linear Programs and Call-by-Value Evaluation

Now we want to use the blinding transformation to examine properties of programs satisfying the P-criterion (Theorem 28).

5.1 Definitions and Main Property

Definition 35 (Linearity). *Let $\mathtt{f}$ be a program terminating by a RPO and $\mathtt{g}$ be a function symbol in $\mathtt{f}$. We say $\mathtt{g}$ is linear in $\mathtt{f}$ if, in the r.h.s. term of any equation for $\mathtt{g}$, there is at most one occurrence of a function symbol $\mathtt{h}$ with same precedence as $\mathtt{g}$. The program $\mathtt{f}$ is linear if all its function symbols are linear.*

Theorem 36. *Let $\mathtt{f}$ be a (possibly non deterministic) program which i) terminates by PPO, ii) admits a quasi-interpretation, iii) is linear. Then $\mathtt{f}$ is strongly polynomial.*

Note that the differences with the P-criterion Theorem from [BMM] (Theorem 28) are that: the program here needs not be deterministic, but linearity is assumed for all function symbols. As a result the bound holds not only for the memoisation semantics, but for the plain call-by-value semantics (and for all execution sequences).

Proof. The quasi-interpretation provides a bound on the size of active judgements via Corollary 27. Linearity of the program ensures that the set of descendants of $\eta = \langle \mathtt{f}, v_1, \cdots, v_n \rangle$ in a call tree with the same precedence as $\mathtt{f}$ is a branch, that is has size bounded by its depth. Termination by PPO ensure that if $\eta' = \langle \mathtt{g}, u_1, \cdots, u_m \rangle$ is the child of η with $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}$ then $|u_i| \leq |v_i|$ and there is at least one j such that $|u_j| < |v_j|$. So, the number of descendants of η with the same precedence is bounded by $\sum |v_i|$. This bounds the number of active judgements by rank.

So we can now use Propositions 21 and 8 and conclude that the size (number of rules) of any derivation $\pi : t \downarrow v$ is polynomially bounded by $|t|$.

Proposition 37. *Let* $\mathtt{f}$ *be a (possibly non deterministic) program which i) terminates by PPO, ii) admits a uniform quasi-interpretation, iii) is linear. Then* $\mathtt{f}$ *is blindly polynomial.*

Proof. Note that:

- $\mathtt{f}$ terminates by PPO, so $\overline{\mathtt{f}}$ also, by Lemma 31;
- the quasi-interpretation for $\mathtt{f}$ is uniform, so $\overline{\mathtt{f}}$ admits a quasi-interpretation, by Prop. 34;
- $\mathtt{f}$ is linear, so $\overline{\mathtt{f}}$ is also linear.

So by Theorem 36 we deduce that $\overline{\mathtt{f}}$ is strongly polynomial. Therefore $\mathtt{f}$ is blindly polynomial.

5.2 Bellantoni-Cook Programs

Let BC the class of Bellantoni-Cook programs, as defined in [BC92] written in a Term Rewriting System framework as in [Moy03]. Function arguments are separated into either *safe* or *normal* arguments, recurrences can only occur over normal arguments and their result can only be used in a safe position. We use here a semi-colon to distinguish between normal (on the left) and safe (on the right) parameters.

Definition 38 (Bellantoni-Cook programs). *The class BC is the smallest class of programs containing:*

- ***(Constant)*** $\mathbf{0}$
- ***(Successors)*** $\mathbf{s}_i(x), i \in \{0, 1\}$

initial functions:

- ***(Projection)*** $\pi_j^{n,m}(x_1, \cdots, x_n; x_{n+1}, \cdots, x_{n+m}) \to x_j$
- ***(Predecessor)*** $\mathtt{p}(;\mathbf{0}) \to \mathbf{0} \quad p(;\mathbf{s}_i(x)) \to x$
- ***(Conditional)*** $\mathtt{C}(;\mathbf{0}, x, y) \to x \quad \mathtt{C}(;\mathbf{s}_0, x, y) \to x \quad \mathtt{C}(;\mathbf{s}_1, x, y) \to y$

and is closed by:

- ***(Safe recursion)***

$$\begin{aligned} \mathtt{f}(\mathbf{0}, x_1, \cdots, x_n; y_1, \cdots, y_m) &\to \mathtt{g}(x_1, \cdots, x_n; y_1, \cdots, y_m) \\ \mathtt{f}(\mathbf{s}_i(z), x_1, \cdots, x_n; y_1, \cdots, y_m) &\to \mathtt{h}_i(z, x_1, \cdots, x_n; y_1, \cdots, y_m, \\ &\qquad \mathtt{f}(z, x_1, \cdots, x_n; y_1, \cdots, y_m)), i \in \{0, 1\} \end{aligned}$$

with $\mathtt{g}, \mathtt{h}_i \in$ *BC (previously defined) ;*

- *(**Safe composition**)*

$$\mathtt{f}(x_1,\cdots,x_n;y_1,\cdots,y_m) \to \mathtt{g}(\mathtt{h}_1(x_1,\cdots,x_n),\ldots,\mathtt{h}_p(x_1,\cdots,x_n);$$
$$\mathtt{l}_1(x_1,\cdots,x_n;y_1,\cdots,y_m),\ldots,\mathtt{l}_q(x_1,\cdots,x_n;y_1,\cdots,y_m))$$

with $\mathtt{g},\mathtt{h}_i,\mathtt{l}_j \in BC$ *;*

It is easy to see that any BC program terminates by PPO and is linear.

Definition 39 (Quasi-interpretations for BC-programs). *A BC-program admits the following quasi-interpretation:*

- $(\!|\mathbf{0}|\!) = 1$ *;*
- $(\!|\mathbf{s}_i|\!)(X) = X + 1$ *;*
- $(\!|\pi|\!)(X_1,\cdots,X_{n+m}) = \max(X_1,\cdots,X_{n+m})$ *;*
- $(\!|\boldsymbol{p}|\!)(X) = X$ *;*
- $(\!|\mathcal{C}|\!)(X,Y,Z) = \max(X,Y,Z)$ *;*

For functions defined by safe recursion of composition, $(\!|\mathtt{f}|\!)(X_1,\cdots,X_n;Y_1,\cdots,Y_m) = q_{\mathtt{f}}(X_1,\cdots,X_n) + \max(Y_1,\cdots,Y_m)$ *with* $q_{\mathtt{f}}$ *defined as follows:*

- $q_{\mathtt{f}}(A,X_1,\cdots,X_n) = A(q_{\mathtt{h}_0}(A,X_1,\cdots,X_n)+q_{\mathtt{h}_1}(A,X_1,\cdots,X_n))+q_{\mathtt{g}}(X_1,\cdots,X_n)$ *if* $\mathtt{f}$ *is defined by safe recursion ;*
- $q_{\mathtt{f}}(X_1,\cdots,X_n) = q_{\mathtt{g}}(q_{\mathtt{h}_1}(X_1,\cdots,X_n),\ldots,q_{\mathtt{h}_p}(X_1,\cdots,X_n))+\sum_i q_{\mathtt{l}_i}(X_1,\cdots,X_n)$ *if* $\mathtt{f}$ *is defined by safe composition.*

Theorem 40. *If* $\mathtt{f}$ *is a program of BC, then* $\mathtt{f}$ *is blindly polynomial.*

Proof. It is sufficient to observe that if f is a BC program, then it is linear and terminates by PPO, and the quasi-interpretation given above is uniform. Therefore by Proposition 37, f is blindly polynomial.

6 Semi-lattices of Quasi-Interpretations

The study of necessary conditions on programs satisfying the P-criterion has drawn our attention to uniform quasi-interpretations. This suggests to consider quasi-interpretations with fixed assignments for constructors and to examine their properties as a class.

Definition 41 (Compatible assignments). *Let* $\mathtt{f}$ *be a program and* $(\!|.|\!)_1$, $(\!|.|\!)_2$ *be two assignments for* $\mathtt{f}$*. We say that they are compatible if for any constructor symbol* $\mathbf{c}$ *we have:*

$$(\!|\mathbf{c}|\!)_1 = (\!|\mathbf{c}|\!)_2.$$

. A family of assignments for $\mathtt{f}$ *is compatible if its elements are pairwise compatible. We use these same definitions for quasi-interpretations.*

Each choice of assignments for constructors thus defines a *maximal compatible family* of quasi-interpretations for a program $\mathtt{f}$: all quasi-interpretations for $\mathtt{f}$ which take these values on $\mathcal{C}$.

We consider on assignments the extensional order $\leq$:

$$(\!|.|\!)_1 \leq (\!|.|\!)_2 \quad \text{iff} \quad \forall f \in \mathcal{C} \cup \mathcal{F}, \forall \boldsymbol{x} \in (\mathbb{R}^+)^k, \qquad (\!|f|\!)_1(\boldsymbol{x}) \leq (\!|f|\!)_2(\boldsymbol{x}).$$

Given two compatible assignments $(\!|.|\!)_1$, $(\!|.|\!)_2$ we denote by $(\!|.|\!)_1 \wedge (\!|.|\!)_2$ the assignment $(\!|.|\!)_0$ defined by:

$$\forall c \in \mathcal{C}. (\!|c|\!)_0 = (\!|c|\!)_1 = (\!|c|\!)_2$$
$$\forall f \in \mathcal{F}. (\!|f|\!)_0 = (\!|f|\!)_1 \wedge (\!|f|\!)_2$$

where $\alpha \wedge \beta$ denotes the greatest lower bound of $\{\alpha, \beta\}$ in the pointwise order. Then we have:

Proposition 42. *Let* `f` *be a program and* $(\!|.|\!)_1$, $(\!|.|\!)_2$ *be two quasi-interpretations for it, then* $(\!|.|\!)_1 \wedge (\!|.|\!)_2$ *is also a quasi-interpretation for* `f`.

To establish this Proposition we need intermediary Lemmas. We continue to denote $(\!|.|\!)_0 = (\!|.|\!)_1 \wedge (\!|.|\!)_2$:

Lemma 43. *For any f of $\mathcal{F}$ we have that $(\!|f|\!)_0$ is monotone and satisfies the subterm property.*

Proof. To prove monotonicity, assume $\boldsymbol{x} \leq \boldsymbol{y}$, for the product ordering. Then, for $i = 1$ or 2: $(\!|f|\!)_0(\boldsymbol{x}) = (\!|f|\!)_1(\boldsymbol{x}) \wedge (\!|f|\!)_2(\boldsymbol{x}) \leq (\!|f|\!)_i(\boldsymbol{x}) \leq (\!|f|\!)_i(\boldsymbol{y})$, using monotonicity of $(\!|f|\!)_i$. As this is true for $i = 1$ and 2 we thus have: $(\!|f|\!)_0(\boldsymbol{x}) \leq (\!|f|\!)_1(\boldsymbol{y}) \wedge (\!|f|\!)_2(\boldsymbol{y}) = (\!|f|\!)_0(\boldsymbol{y})$. It is also easy to check that $(\!|f|\!)_0$ satisfies the subterm property.

Lemma 44. *Let t be a term. We have: $(\!|t|\!)_0 \leq (\!|t|\!)_i$, for $i = 1, 2$.*

Proof. By induction on t, using the definition of $(\!|f|\!)_0$ and $(\!|c|\!)_0$, and the monotonicity property of Lemma 43.

Lemma 45. *Let $g(p_1, \cdots, p_n) \to t$ be an equation of the program f. We have:*

$$(\!|g|\!)_0 \circ ((\!|p_1|\!)_0, \ldots, (\!|p_n|\!)_0) \geq (\!|t|\!)_0.$$

Proof. As patterns only contain constructor and variable symbols, and by definition of $(\!|.|\!)_0$, if p is a pattern we have: $(\!|p|\!)_0 = (\!|p|\!)_1 = (\!|p|\!)_2$. Let $i = 1$ or 2; we have:

$$\begin{aligned}(\!|g|\!)_i((\!|p_1|\!)_i(\boldsymbol{x}), \ldots, (\!|p_n|\!)_i(\boldsymbol{x})) &\geq (\!|t|\!)_i(\boldsymbol{x}) && \text{because } (\!|.|\!)_i \text{ is a quasi-interpretation,}\\ &\geq (\!|t|\!)_0(\boldsymbol{x}) && \text{with Lemma 44.}\end{aligned}$$

So:

$$(\!|g|\!)_i((\!|p_1|\!)_0(\boldsymbol{x}), \ldots, (\!|p_n|\!)_0(\boldsymbol{x})) \geq (\!|t|\!)_0(\boldsymbol{x}), \text{ as } (\!|p_j|\!)_i = (\!|p_j|\!)_0.$$

Write $\boldsymbol{y} = ((\!|p_1|\!)_0(\boldsymbol{x}), \ldots, (\!|p_n|\!)_0(\boldsymbol{x}))$. As $(\!|g|\!)_1(\boldsymbol{y}) \geq (\!|t|\!)_0(\boldsymbol{x})$ and $(\!|g|\!)_2(\boldsymbol{y}) \geq (\!|t|\!)_0(\boldsymbol{x})$, by definition of $(\!|g|\!)_0$ we get $(\!|g|\!)_0(\boldsymbol{y}) \geq (\!|t|\!)_0(\boldsymbol{x})$, which ends the proof.

Now we can proceed with the proof of Prop. 42:

Proof (Proof of Prop. 42). Observe that Lemma 43 ensures that $(\!|.|\!)_0$ satisfies the monotonicity and the subterm conditions, and Lemma 45 that it satisfies the condition w.r.t. the equations of the program. The conditions for the constructors are also satisfied by definition. Therefore $(\!|.|\!)_0$ is a quasi-interpretation.

Proposition 46. *Let* $\mathtt{f}$ *be a program and* $\mathcal{Q}$ *be a family of compatible quasi-interpretations for* $\mathtt{f}$*, then* $\wedge_{(\!|.|\!)\in\mathcal{Q}}(\!|.|\!)$ *is a quasi-interpretation for* $\mathtt{f}$*. Therefore maximal compatible families of quasi-interpretations for* $\mathtt{f}$ *have an inferior semi-lattice structure for* $\leq$*.*

Proof. It is sufficient to generalize Lemmas 43 and 45 to the case of an arbitrary family $\mathcal{Q}$ and to apply the same argument as for the proof of Prop. 42.

7 Extending the P-criterion

Blind abstraction suggest to consider not only the PPO ordering from the P-criterion, but also an extension which is invariant by the blinding map, the EPPO ordering (see Subsection 4.3). It is thus natural to ask whether EPPO enjoys the same property as PPO. We prove in this section that with EPPO we can still bound the size of the call-dag and thus generalize the P-criterion. Then, we will also consider the *bounded value property* which is an extension of the notion of QI. Here, we bound the number of nodes in the call-dag with a given precedence. Then, Prop. 21 bounds the total number of nodes in the call-dag.

Fact 3 *Since we're working over words (unary constructors), patterns are either constructors terms (that is, words), or have the form* $p = \mathbf{s}_1(\mathbf{s}_2 \dots \mathbf{s}_n(x)\dots)$*. In the second case, we will write* $p = \Omega(x)$ *with* $\Omega = \mathbf{s}_1\mathbf{s}_2\dots\mathbf{s}_n$*.*

The length of a pattern is the length of the corresponding word: $|p| = |\Omega|$

Proposition 47. *In a program terminating by PPO or EPPO, the only calls at the same precedence that can occur are of the form*

$$\mathtt{f}(p_1, \cdots, p_n) \to C[\mathtt{g}_1(q_1^1, \cdots, q_m^1), \dots, \mathtt{g}_p(q_1^p, \cdots, q_l^p)]$$

where $C[.]$ *is some context,* $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}^k$ *and* p_i, q_j^k *are patterns. Moreover, each variable appearing in a* q_i^k *appears in* p_i*.*

Proof. This is a direct consequence of the termination ordering.

Since we will only consider individual calls, we will put in the context all but one of the $\mathtt{g}_k$: $\mathtt{f}(p_1, \cdots, p_n) \to C[\mathtt{g}(q_1, \cdots, q_m)]$

Definition 48 (Production size). *Let* $\mathtt{f}$ *be a program terminating by EPPO. Let* $\mathtt{f}(p_1, \cdots, p_n) \to C[\mathtt{g}(q_1, \cdots, q_m)]$ *be a call in it where* $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}$*. The* production size *of this call is* $\max_i\{|q_i|\}$*. The production size of an equation is the greatest production size of any call (at the same precedence) in it. That is if we have an equation* $e = \mathtt{f}(p_1, \cdots, p_n) \to C[\mathtt{g}_1(q_1^1, \cdots, q_m^1), \dots, \mathtt{g}_p(q_1^p, \cdots, q_l^p)]$ *where* $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}_k$ *then its production size is* $K_e = \max|q_j^k|$ *The production size of a function symbol is the maximum production size of any equation defining a function with the same precedence:* $K_{\mathtt{h}} = \max_{\mathtt{g}\approx_{\mathcal{F}}\mathtt{h}} \max_{e=\mathtt{g}(\dots)\to r} K_e$

Definition 49 (Normality). *Let $\mathtt{f}$ be a program. A function symbol $\mathtt{h}$ in it is* normal *if the patterns in the definitions of functions with the same precedence are bigger than its production size:*

$$\forall \mathtt{g} \approx_{\mathcal{F}} \mathtt{h}, \forall \mathtt{g}(q_1, \cdots, q_m) \to r \in \mathcal{E}, |q_i| \geq K_{\mathtt{h}}$$

Let $\mathtt{f}$ be a program. It is normal *if all function symbols in it are normal.*

This means that during recursive calls, every constructor produced at a given moment will be consumed by the following pattern matching.

Lemma 50. *A EPPO-program can be normalised with an exponential growth* in the size of the program.

The exponential is in the difference between the size of the biggest production and the size of the smallest pattern (with respect to each precedence).

Proof (Sketch). The idea is to extend the small pattern matchings so that their length reaches the length of the biggest production. This is illustrated by the following example:

$$\begin{aligned} \mathtt{f}(\mathbf{s}_1(\mathbf{s}_1(\mathbf{s}_1(x)))) &\to \mathtt{f}(\mathbf{s}_0(\mathbf{s}_0(x))) \\ \mathtt{f}(\mathbf{s}_0(x)) &\to \mathtt{f}(x) \end{aligned}$$

In this case, the biggest production has size 2 but the shortest pattern matching has only size 1. We can normalise the program as follows:

$$\begin{aligned} \mathtt{f}(\mathbf{s}_1(\mathbf{s}_1(\mathbf{s}_1(x)))) &\to \mathtt{f}(\mathbf{s}_0(\mathbf{s}_0(x))) \\ \mathtt{f}(\mathbf{s}_0(\mathbf{s}_0(x))) &\to \mathtt{f}(\mathbf{s}_0(x)) \\ \mathtt{f}(\mathbf{s}_0(\mathbf{s}_1(x))) &\to \mathtt{f}(\mathbf{s}_1(x)) \end{aligned}$$

Even if the process does extend productions as well as patterns, it does terminate because only the smallest patterns, hence the smallest productions (due to termination ordering) are extended this way.

Notice that the exponential growth is indeed in the initial size of the program and does not depend on the size of any input. Since the size of the call-dag is bounded by the size of the inputs, this does not hamper the polynomial bound. The normalization process preserves termination by PPO and EPPO, semantics and does not decrease time complexity. Hence, bounding the time complexity of the normalized program is sufficient to bound the time complexity of the initial program. In the following, we only consider normal programs.

Let $\mathtt{f}$ be a program and $\mathtt{g}$ be a function, we will enumerate all the symbol of same precedence as $\mathtt{g}$ in the rhs of $\mathtt{f}$ and label them $\mathtt{g}^1, \ldots, \mathtt{g}^n$. This is simply an enumeration, not a renaming of the symbols and if a given symbol appears several times (in several equations or in the same one), it will be given several labels (one for each occurrence) with this enumeration. Now, a path in the call-dag staying only at the same precedence as $\mathtt{g}$ is canonically identified by a word over $\{\mathtt{g}^1, \ldots, \mathtt{g}^n\}$. We write $\eta \overset{\omega}{\rightsquigarrow} \eta'$ to denote that η is an ancestor of η' and ω is the path between them.

Lemma 51. *Let $\eta_1 = \langle \mathtt{f}, v_1, \cdots, v_n \rangle$ and $\eta_2 = \langle \mathtt{g}, u_1, \cdots, u_m \rangle$ be two states such that $\eta_1 \overset{e}{\rightsquigarrow} \eta_2$ and both function symbols have the same precedence. Then $|v_i| \geq |u_i|$ for all i and there exists j such that $|v_j| > |u_j|$.*

Proof. This is a consequence of the termination proof by EPPO.

Corollary 52. *Let $\eta = \langle \mathtt{f}, v_1, \cdots, v_n \rangle$ be a state. Any branch in the call-dag starting from η has at most $n \times (\max |v_i|)$ nodes with the same precedence as $\mathtt{f}$.*

Lemma 53. *Suppose that we have labels α, β and γ and nodes such that $\eta \overset{\alpha}{\rightsquigarrow} \eta_1 \overset{\beta}{\rightsquigarrow} \eta_1' \overset{\gamma}{\rightsquigarrow} \eta_1''$ and $\eta \overset{\beta}{\rightsquigarrow} \eta_2 \overset{\alpha}{\rightsquigarrow} \eta_2' \overset{\gamma}{\rightsquigarrow} \eta_2''$. Then $\eta_1'' = \eta_2''$.*

Proof. Since labels are unique, the function symbols in η_1'' and η_2'' are the same. It is sufficient to show that the ith components are the same and apply the same argument for the other parameters.

Let $q, v, v', v'', u, u', u''$ be the ith parameters of $\eta, \eta_1, \eta_1', \eta_1'', \eta_2, \eta_2', \eta_2''$ respectively. Since we're working on words, an equation e has, with respect to the ith parameter, the form:

$$\mathtt{f}(\ldots, \Omega_e(x), \ldots) \to C[\mathtt{g}(\ldots, \Omega_e'(x), \ldots)]$$

and normalization implies that $|\Omega_e'| \leq |\Omega_e|$ (previous lemma).

So, in our case, we have:

$$\begin{aligned} q &= \Omega_\alpha(x) \to \Omega_\alpha'(x) = v = \Omega_\beta(x') \to \Omega_\beta'(x') \\ &= v' = \Omega_\gamma(x'') \to \Omega_\gamma'(x'') = v'' \\ q &= \Omega_\beta(y) \to \Omega_\beta'(y) = u = \Omega_\alpha(y') \to \Omega_\alpha'(y') \\ &= u' = \Omega_\gamma(y'') \to \Omega_\gamma'(y'') = u'' \end{aligned}$$

Because of normalization, $|\Omega_\beta| \geq |\Omega_\alpha'|$. Hence x' is a suffix of x, itself a suffix of q. Similarly, x'' and y'' are suffixes of q.

Since they're both suffixes of the same word, it is sufficient to show that they have the same length in order to show that they are identical.

$$\begin{aligned} |x| &= |q| - |\Omega_\alpha| \\ |x'| &= |v| - |\Omega_\beta| = |q| - |\Omega_\alpha| + |\Omega_\alpha'| - |\Omega_\beta| \\ |x''| &= |q| - |\Omega_\alpha| + |\Omega_\alpha'| - |\Omega_\beta| + |\Omega_\beta'| - |\Omega_\gamma| \\ |y''| &= |q| - |\Omega_\beta| + |\Omega_\beta'| - |\Omega_\alpha| + |\Omega_\alpha'| - |\Omega_\gamma| \end{aligned}$$

So $x'' = y''$ and thus $v'' = u''$.

Corollary 54. *Let ω_1, ω_2 be words and α be a label such that: $\eta \overset{\omega_1}{\rightsquigarrow} \eta_1 \overset{\alpha}{\rightsquigarrow} \eta_1'$ and $\eta \overset{\omega_2}{\rightsquigarrow} \eta_2 \overset{\alpha}{\rightsquigarrow} \eta_2'$. If ω_1 and ω_2 have the same commutative image, then $\eta_1' = \eta_2'$.*

Proof. This is a generalization of the previous proof, not an induction on it. If $\omega_1 = \alpha_1 \ldots \alpha_n$ then the size in the last node is:

$$|x'| = |q| - \sum |\Omega_{\alpha_i}| + \sum |\Omega_{\alpha_i}'| - |\Omega_\alpha|$$

which is only dependent on the commutative image of ω_1.

So, when using the semantics with memoisation, the number of nodes (at a given precedence) in the call-dag is roughly equal to the number of paths *modulo commutativity*. So any path can be associated with the vector whose components are the number of occurrences of the corresponding label in it.

Proposition 55. *Let $\mathcal{T}$ be a call-dag and $\eta = \langle \mathtt{f}, v_1, \cdots, v_n \rangle$ be a node in it. Let $I = n \times (\max |v_j|)$ and M be the number of functions with the same precedence as $\mathtt{f}$. The number of descendants of η in $\mathcal{T}$ with the same precedence as $\mathtt{f}$ is bounded by $(I+1)^M$, that is a polynomial in $|\eta|$.*

Proof. Any descendant of η with the same precedence can be identified with a word over $\{\mathtt{f}^1, \ldots, \mathtt{f}^M\}$. By Corollary 52 we know that these words have length at most I and by Corollary 54 we know that it is sufficient to consider these words modulo commutativity.

Modulo commutativity, words can be identified to vectors with as many components as the number of letters in the alphabet and whose sum of components is equal to the length of the word.

Let D_i^n be the number of elements from $\mathbb{N}^n$ whose sum is i. This is the number of words of length i over a n-ary alphabet modulo commutativity.

Clearly, $D_{i-1}^n \leq D_i^n$ (take all the n-uple whose sum is $i-1$, add 1 to the first component, you obtain D_{i-1}^n different n-uple whose sum of components is i).

Now, to count D_i^n, proceed as follows: Choose a value j for the first component, then you have to find $n-1$ components whose sum is $i-j$, there are D_{i-j}^{n-1} such elements.

$$D_i^n = \sum_{0 \leq j \leq i} D_{i-j}^{n-1} = \sum_{0 \leq j \leq i} D_j^{n-1} \leq (i+1) \times D_i^{n-1} \leq (i+1)^{n-1} \times D_i^1 \leq (i+1)^n$$

Definition 56 (Bounded Values). *A program $\mathtt{f} = (\mathcal{X}, \mathcal{C}, \mathcal{F}, \mathcal{E})$ has polynomially bounded values iff for every function symbol $\mathtt{g} \in \mathcal{F}$, there is a polynomial $p_{\mathtt{g}} : \mathbb{N} \to \mathbb{N}$ such that for every state η' appearing in a call tree for $\eta = (\mathtt{g}, v_1, \ldots, v_n)$, $|\eta'| \leq p_{\mathtt{g}}(|\eta|)$.*

Theorem 57. *The set of functions computed by programs who terminate by EPPO and have bounded value property is exactly* Ptime.

Proof. Proposition 55 bounds the size of the call dag by rank. Bounded value property bounds the size of nodes in the call dag. So we can apply Proposition 11 to bound the size of any derivation. The converse is obtained from the P-criterion.

Theorem 58. *Let f be a deterministic program terminating by EPPO. Then the following two conditions are equivalent:*
1. *f has polynomially bounded values;*
2. *f is polytime in the call-by-value semantics with memoisation.*

Proof. The implication $1 \Rightarrow 2$ is proved as follows: Termination by EPPO provides a polynomial bound on the size of the call-dag (by rank) via Proposition 55

and the bounded values property provides a polynomial bound on the size of nodes in the call dag. Hence we can apply Proposition 11 and bound the size of any derivation $\pi : \langle \emptyset, t \rangle \Downarrow \langle C, v \rangle$ by $P(|t|)$ for some polynomial P.

For the converse, it is sufficient to see that a state appearing in the call dag also appear in the final cache. Since the size of any term in the cache is bounded by the size of the proof (because we need to perform as many (Constructor) rules as needed to construct the term), it is polynomially bounded because the program is polytime.

8 Conclusions

In this paper, blind abstractions of first-order functional programs have been introduced and exploited in understanding the intensional expressive power of quasi-interpretations. In particular, being blindly polytime has been proved to be a necessary condition in presence of linear programs, product path-orderings and uniform quasi-interpretations. This study has lead us to some other interesting results about the lattice-structure of uniform quasi-interpretations and the possibility of extending product path-orderings preserving polytime soundness.

Further work includes investigations on conditions characterizing the class of programs (or proofs) captured by quasi-interpretation. In particular, it is still open whether being blindly polytime is a necessary *and sufficient* conditions for a program to have a quasi-interpretation (provided some sort of path-ordering for it exists).

References

[Ama05] R. Amadio. Synthesis of max-plus quasi-interpretations. *Fundamenta Informaticae*, 65:29–60, 2005.

[BC92] Stephen J. Bellantoni and Stephen A. Cook. A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity*, 2:97–110, 1992.

[BMM] G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. Quasi-interpretation: a way to control resources. *Theoretical Computer Science*. Under revision.

[BMM05] G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. Quasi-Interpretations and Small Space Bounds. In J. Giesl, editor, *Rewriting Techniques and Applications*, volume 3467 of *Lecture Notes in Computer Science*. Springer, April 2005.

[BMP06] G. Bonfante, J.-Y. Marion, and R. Péchoux. On the modularity of quasi-interpretations. Technical report, LORIA, 2006. Preprint.

[Der82] Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982.

[DP02] N. Danner and C. Pollett. Minimization and NP multifunctions. *Theoretical Computer Science*, 2002. To appear.

[Gir98] J.-Y. Girard. Light linear logic. *Information and Computation*, 143:175–204, 1998.

[Hof99] Martin Hofmann. Linear types and Non-Size Increasing polynomial time computation. In *Proceedings of the Fourteenth IEEE Symposium on Logic in Computer Science (LICS'99)*, pages 464–473, 1999.

[Hue80] Gérard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.

[KL80] S. Kamin and J.-J. Lévy. Attempts for generalising the recursive path orderings. Technical report, Univerité de l'Illinois, Urbana, 1980. unpublished note. Accessible on `http://www.ens-lyon.fr/pierre.lescanne/not_accessible.html`.

[Lei94] D. Leivant. Predicative recurrence and computational complexity I: word recurrence and poly-time. In *Feasible Mathematics II*, pages 320–343. Birkhauser, 1994.

[MM00] Jean-Yves Marion and Jean-Yves Moyen. Efficient First Order Functional Program Interpreter with Time Bound Certifications. In Michel Parigot and Andrei Voronkov, editors, *LPAR*, volume 1955 of *Lecture Notes in Artificial Intelligence*, pages 25–42. Springer, November 2000.

[Moy03] J.-Y. Moyen. *Analyse de la complexité et transformation de programmes.* Thèse d'université, Nancy 2, Dec 2003.